Submitted to the Technical Communications of the International Conference on Logic Programming (ICLP'10)
http://www.floc-conference.org/ICLP-home.html

IMPROVING THE EFFICIENCY OF GIBBS SAMPLING FOR PROBABILISTIC LOGICAL MODELS BY MEANS OF PROGRAM SPECIALIZATION

DAAN FIERENS

Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, 3001 Heverlee, Belgium
E-mail address: Daan.Fierens@cs.kuleuven.be

Abstract. There is currently a large interest in probabilistic logical models. A popular algorithm for approximate probabilistic inference with such models is Gibbs sampling. From a computational perspective, Gibbs sampling boils down to repeatedly executing certain queries on a knowledge base composed of a static part and a dynamic part. The larger the static part, the more redundancy there is in these repeated calls. This is problematic since inefficient Gibbs sampling yields poor approximations.
We show how to apply program specialization to make Gibbs sampling more efficient. Concretely, we develop an algorithm that specializes the definitions of the query-predicates with respect to the static part of the knowledge base. In experiments on real-world benchmarks we obtain speedups of up to an order of magnitude.

1. Introduction

In the field of artificial intelligence there is a large interest in *probabilistic logical models* (probabilistic extensions of logic programs and first-order logical extensions of probabilistic models such as Bayesian networks) [3, 10, 5]. *Probabilistic inference* with such a model is the task of answering various questions about the probability distribution specified by the model, usually conditioned on certain observations (the *evidence*). A variety of inference algorithms is being used. A popular algorithm for approximate probabilistic inference is *Gibbs sampling* [2, 11]. Gibbs sampling works by drawing samples from the considered probability distribution conditioned on the evidence. These samples can be used to compute an approximate answer to the probabilistic questions of interest. It is important that the process of drawing samples is efficient because the more samples can be drawn per time-unit, the more accurate the answers will be (i.e., the closer to the correct answer).

Computationally, Gibbs sampling boils down to repeatedly executing the same queries on a knowledge base composed of a static part (the evidence and background knowledge) and a highly dynamic part that changes at runtime because of the sampling. The more evidence, the larger the static part of the knowledge base, so the more redundancy there

1998 ACM Subject Classification: I.2.2, G.3, D.1.6.
Key words and phrases: probabilistic logical models, probabilistic logic programming, program specialization, Gibbs sampling.
This research is supported by Research Foundation-Flanders (FWO Vlaanderen), GOA/08/008 'Probabilistic Logic Learning' and Research Fund K.U.Leuven.

is in these repeated calls. Since it is important that the sampling process is efficient, this redundancy needs to be reduced as much as possible. In this paper we show how to do this by applying *program specialization* to the definitions of the query-predicates: we specialize these definitions with respect to the static part of the knowledge base. While a lot of work about logic program specialization is about exploiting static information about the input arguments of queries (partial deduction [6]), we instead exploit static information about the knowledge base on which the queries are executed.

While the above applies to all kinds of probabilistic logical models and programs, we will focus on models that are first-order logical or "relational" extensions of Bayesian networks [3, 5]. Concretely, we use the general framework of *parameterized Bayesian networks* [10].

The *contributions of this paper* are the following. First, we show how to represent parameterized Bayesian networks in Prolog (Section 3). Second, we show how to implement Gibbs sampling in Prolog and show that doing this efficiently poses challenges from the logic programming point of view (Section 4). Third, we develop an algorithm for specializing the considered logic programs with respect to the evidence (Section 5). Fourth, we perform experiments on real-world benchmarks to investigate the influence of specialization on the efficiency of Gibbs sampling. Our results show that specialization yields speedups of up to an order of magnitude and that these speedups grow with the data-size (Section 6). The latter two are the main contributions of this paper, the first two are minor contributions.

We first give some background on probability theory and Bayesian networks.

2. Preliminaries: Probability Theory and Bayesian Networks

In probability theory [8] one models the world in terms of *random variables (RVs)*. Each state of the world corresponds to a joint state of all considered RVs. We use upper case letters to denote single RVs and boldface upper case letters to denote sets of RVs. We refer to the set of possible states of an RV X (i.e. the set of values that X can take) as the *range* of X, denoted $range(X)$. We consider only *discrete RVs*, i.e. RVs with a finite range.

A *probability distribution* on a finite set S is a function that maps each $x \in S$ to a number $P(x) \in [0, 1]$ such that $\sum_{x \in S} P(x) = 1$. A probability distribution for an RV X is a probability distribution on the set $range(X)$. A *conditional probability distribution (CPD)* for an RV X conditioned on a set of other RVs $\mathbf{Y}$ is a function that maps each possible joint state of $\mathbf{Y}$ to a probability distribution for X.

Syntactically, a *Bayesian network* [8] for a set of RVs $\mathbf{X}$ is a set of CPDs: for each $X \in \mathbf{X}$ there is one CPD for X conditioned on a (possibly empty) set of RVs called the *parents* of X. Intuitively, the CPD for X specifies the direct probabilistic influence of X's parents on X. The probability distribution for X conditioned on its parents $\mathbf{pa}(X)$, as determined by the CPD for X, is denoted $P(X \mid \mathbf{pa}(X))$.

Semantically, a Bayesian network represents a probability distribution $P(\mathbf{X})$ on the set of all possible joint states of $\mathbf{X}$. Concretely, $P(\mathbf{X})$ is the product of all the CPDs in the Bayesian network: $P(\mathbf{X}) = \prod_{X \in \mathbf{X}} P(X \mid \mathbf{pa}(X))$. It can be shown that $P(\mathbf{X})$ is a proper probability distribution provided that the parent relation is acyclic (the parent relation is often visualized as a directed acyclic graph but given the CPDs this graph is redundant).

3. Parameterized Bayesian Networks

Bayesian networks essentially use a propositional representation. Several ways of lifting them to a first-order representation have been proposed [3, Ch.6,7,13] [5]. There also exist several probabilistic extensions of logic programming, such as PRISM, Independent Choice Logic and ProbLog [3, Ch.5,8]. Both kinds of probabilistic logical models (probabilistic logic programs and the extensions of Bayesian networks) essentially serve the same purpose. In this paper we focus on the Bayesian network approach. Our main motivation for this choice is that this paper is about Gibbs sampling and this has been well-studied in the context of Bayesian networks. There are many different representation languages for first-order logical or "relational" extensions of Bayesian networks. We use the general framework of *parameterized Bayesian networks* [10]. While this framework is perhaps not a full-fledged knowledge representation language, it does offer a representation that is suited to implement probabilistic inference algorithms on.

We now briefly introduce parameterized Bayesian networks. Like Bayesian networks use RVs, parameterized Bayesian networks use so-called *parameterized RVs* [10]. Parameterized RVs have a number of typed parameters ranging over certain populations. When each parameter in a parameterized RV is instantiated or "grounded" to a particular element of its population we obtain a regular or *"concrete"* RV. To each parameterized RV we associate a *parameterized CPD* (see below) with the same parameters as the parameterized RV.

Syntactically, a parameterized Bayesian network is a set of parameterized CPDs, one for each parameterized RV. Semantically, a parameterized Bayesian network $\mathcal{B}$, in combination with a given population for each type, specifies a probability distribution. Let $\mathbf{X}$ denote the set of all concrete RVs obtained by grounding all parameterized RVs in $\mathcal{B}$ with respect to their populations. The probability distribution specified by $\mathcal{B}$ is the following distribution on the set of all possible joint states of $\mathbf{X}$: $P(\mathbf{X}) = \prod_{X \in \mathbf{X}} P(X \mid \mathbf{pa}(X))$, where $P(X \mid \mathbf{pa}(X))$ denotes the probability distribution for X as determined by its parameterized CPD.

Rather than providing a formal discussion of parameterized Bayesian networks we show how they can be represented in Prolog (as far as we know this has not been done before).

To deal with parameterized RVs in Prolog we associate to each of them a unique predicate: for a parameterized RV with n parameters we use a $(n+1)$-ary predicate, the first n arguments correspond to the parameters, the last argument represents the state of the RV. We refer to these predicates as *state predicates*.

Syntactically a parameterized Bayesian network is a set of parameterized CPDs. To deal with parameterized CPDs we also associate to each of them a unique predicate, the last argument now represents a probability distribution on the range of the associated RV. We refer to these predicates as *CPD-predicates*. In this paper we assume that each CPD-predicate is defined by a decision list. A *decision list* is an ordered set of rules such that there is always at least one rule that applies, and of all rules that apply only the first one fires (in Prolog this is achieved by putting a cut at the end of each body and having a last clause with *true* as the body).

Example 3.1. Consider a university domain. Suppose that we use the following parameterized RVs: level (with a parameter from the population of courses), iq and graduates (each with a student parameter) and grade (with a student parameter and a course parameter). To represent the state of the RVs we use the state predicates $level/2$, $iq/2$, $graduates/2$ and $grade/3$. The meaning of for instance $level/2$ is that the atom $level(C, L)$ is true if the parameterized RV *level* for the course C is in state L .

To represent parameterized CPDs we use CPD-predicates *cpd_level*/2, *cpd_iq*/2, *cpd_grade*/3 and *cpd_graduates*/2. If the *level* RVs for instance do not have parents, their parameterized CPD could be defined as follows.

```
cpd_level(_C,[intro:0.4,advanced:0.6]).
```

Note that we use lists like `[intro:0.4,advanced:0.6]` to represent probability distributions. The other parameterized CPDs could for instance be defined as follows.

```
cpd_iq(_S,[high:0.5,low:0.5]).

cpd_grade(S,C,[a:0.7,b:0.2,c:0.1]) :- iq(S,high), level(C,intro), !.
cpd_grade(S,C,[a:0.2,b:0.2,c:0.6]) :- iq(S,low), level(C,advanced), !.
cpd_grade(S,C,[a:0.3,b:0.4,c:0.3]).

cpd_graduates(S,[yes:0.2,no:0.8]) :- grade(S,_C,c), !.
cpd_graduates(S,[yes:0.5,no:0.5]) :- findall(C,grade(S,C,a),L),
                                                    length(L,N), N<2, !.
cpd_graduates(S,[yes:0.9,no:0.1]).
```

In the bodies of the clauses defining the CPD-predicates we allow the use of state predicates (e.g. *iq*/2 and *level*/2 in the clauses for *cpd_grade*/3) and of background predicates, but not of CPD-predicates. With *background predicates* we mean auxiliary predicates that do not depend on the state of RVs (this includes built-ins such as *length*/2). We assume that the definitions of the background predicates are available in a *background knowledge base*. We also allow the use of meta-predicates (such as *findall*/3) but not of predicates with side-effects (such as *assert/1*).

When we know the population for each type (e.g. we know the set of students and the set of courses) we also know the set of concrete RVs $\mathbf{X}$. Suppose that in addition we also know the state of these concrete RVs because we are given a knowledge base with facts defining the state predicates (e.g. a fact $grade(s1, c1, a)$ indicates that student $s1$ has grade 'a' for course $c1$). We can then obtain the probability distribution for a concrete RV conditioned on its parents by simply calling the associated CPD-predicate on this knowledge base. For instance, we obtain the probability that the student $s1$ will graduate conditioned on her grades by calling $cpd\_graduates(s1, Distribution)$. We refer to this as *calling the CPD for that concrete RV*. Since we represent each parameterized CPD as a decision list it is guaranteed that this always returns exactly one probability distribution.[1]

As we explain in the next section, calling a CPD is an operation that needs to be performed frequently during probabilistic inference. Another such operation is *setting a concrete RV to a given state*. This is done by modifying the corresponding fact in the knowledge base (e.g. the fact $grade(s1, c1, a)$ is turned into $grade(s1, c1, b)$ [4]).

4. Probabilistic Inference with Parameterized Bayesian Networks

Given the population for each type, a parameterized Bayesian network defines a probability distribution $P(\mathbf{X})$ on the set of all possible joint states of the concrete RVs $\mathbf{X}$. In a typical inference scenario, the state of a subset of all these RVs is observed. This information is called the *evidence*. *Probabilistic inference* is the task of answering certain questions

[1]Some CPD-predicates are defined by non-ground facts (e.g. *cpd_level*/2). This does not cause problems because we always call CPD-predicates with all arguments except the last instantiated.

about the probability distribution $P(\mathbf{X})$ conditioned on the evidence. The most common inference task is to compute marginal probabilities. A *marginal probability* is the probability that a particular RV is in a particular state. For instance, given the level of all courses and the grades of all students for all courses (the evidence), we might want to compute for each student the probability that she has a high IQ. In theory such probabilities can be computed by performing a series of sum and product operations on the probability distributions specified by the parameterized CPDs. Unfortunately, for real-world population sizes this is computationally intractable (inference with Bayesian networks is NP-hard [8]). Hence, one often uses *approximate probabilistic inference* instead. An important class of approximate inference algorithms are *Monte Carlo algorithms* that draw samples from the given distribution conditioned on the evidence. Various algorithms are being used, a very popular one is *Gibbs sampling* [2, 11].

Let $\mathbf{O}$ denote the set of all observed concrete RVs (i.e. the RVs for which we have evidence), and $\mathbf{U}$ the set of all unobserved ones ($\mathbf{U} = \mathbf{X} \setminus \mathbf{O}$). Below we assume that we need to compute marginal probabilities for all unobserved RVs. Pseudocode for the Gibbs sampling algorithm is shown in Figure 1. We now explain this further.

```
procedure GIBBS_SAMPLING(O, U)
1 for each O ∈ O
2     set O to its known state
3 for each U ∈ U
4     set U to random state ∈ range(U)
5     initialize all counters for U
6 repeat until enough samples
7     for each U ∈ U
8         RESAMPLE(U)
9 compute estimates from counters
```

```
procedure RESAMPLE(U)
1 call the CPD for U
2 for each u ∈ range(U)
3     set U to state u
4     for each child X of U
5         call the CPD for X
6 calculate P_resample(U)
7 sample u_new from P_resample(U)
8 set U to u_new
9 increment counter for (U, u_new)
```

Figure 1: The Gibbs sampling algorithm (left) and its RESAMPLE procedure (right).

Before the start of the sampling process all observed RVs are instantiated to their known state and all unobserved RVs are instantiated to a random state. In terms of our implementation in Prolog, this is done by creating a knowledge base defining all the state predicates: for each RV $\in \mathbf{O} \cup \mathbf{U}$ there is one fact for the corresponding state predicate. Before we start sampling, we also create a number of counters: for each $U \in \mathbf{U}$ and each $u \in range(U)$ we create a counter to store the number of samples in which U is in state u. All counters are initialized to zero.

Let us now consider the sampling process itself. To create one sample, we visit (in an arbitrary but fixed order) all unobserved RVs. When we visit an RV U, we "resample" it. The idea is to sample the new state from the probability distribution for U conditioned on the current state of *all* other RVs. For details on how to construct this distribution $P_{resample}(U)$ we refer to Bidyuk and Dechter [1], here we focus on the main computations that this requires (see the RESAMPLE procedure in Figure 1): first we need to call the CPD for U, then we loop over all possible states of U and for each state u we set U to u and call the CPDs of each of the children of U.Based on the information returned by all these CPD-calls it is straightforward to construct the distribution $P_{resample}(U)$. We then randomly sample a state from this distribution, set U to this new state and increment the appropriate counter for U.

The above is done for all unobserved RVs, yielding one sample.[2] Note that observed RVs are clamped to their known state, hence the generated sample is guaranteed to be consistent with the evidence. This entire procedure is repeated N times, yielding N samples. It is then straightforward to construct an estimate of all required marginal probabilities based on the computed counts. For instance, the estimated probability that student $s1$ has a high IQ conditioned on the evidence is the number of samples in which the RV iq for $s1$ was in the state 'high', divided by N.

The higher the number of samples N, the closer the estimated marginal probabilities will be to their correct values [1, 4]. Gibbs sampling is often used by giving the sampling process a fixed time to run before computing the estimates. In this case, the less time it takes to draw a single sample, the more samples can be drawn in the given time, so the higher the accuracy of the estimates. In other words: any gain in efficiency of the sampling process might lead to a gain in accuracy of the estimates. Hence it is crucial to implement the sampling process as efficiently as possible.

The Gibbs sampling algorithm uses several operations, but there is one operation that we clearly found to be the computational bottleneck, namely *calling the CPDs*. This operation occurs inside several nested loops (see line 5 of the RESAMPLE procedure in Figure 1) and is hence performed many times. The knowledge base on which these CPD-queries are called is highly dynamic: the state of the unobserved RVs changes continuously because they are being resampled. This is only one part of the knowledge base, however. The part that is about the observed RVs (the evidence) stays constant during the entire sampling process. This static part of the knowledge base causes redundancy in the repeated calls of the CPD-queries since part of the computations are performed over and over again. The more evidence we have, the larger the redundancy. In many practical cases, the amount of evidence is considerable and hence the redundancy can be large. Since we want the sampling process to be as efficient as possible, this redundancy needs to be removed. In the next section we show how this can be done by means of program specialization.

5. Applying Logic Program Specialization to Parameterized CPDs

The idea is to *specialize the definitions of the CPD-predicates with respect to the static part of the knowledge base*. Recall that we define each CPD-predicate in Prolog by means of a decision list (Example 3.1). Our specialization approach is a source-to-source transformation that takes three inputs: 1) the decision lists for all the CPD-predicates, 2) the evidence (i.e. the observed RVs with their observed states), and 3) the background knowledge base. The output of the transformation is a specialized version of the decision lists. The transformation is such that Gibbs sampling produces exactly the same sequence of samples with the specialized decision lists as with the original ones (but in a more efficient way).

We use the term *CPD-query* to refer to any atom for a CPD-predicate with the last argument uninstantiated and all other arguments instantiated to elements of the proper populations. For instance, $cpd\_grade(s, c, Distribution)$ is a CPD-query if s is in the considered population of students and c in the population of courses. All calls to CPD-predicates that occur during Gibbs sampling are calls of CPD-queries. Moreover, there is only a fixed set of CPD-queries that are ever called during Gibbs sampling: by examining the RESAMPLE procedure (Figure 1) one can see that the only CPD-queries that are ever called are those

[2] In practice we use a slight variation of this procedure which includes a number of common optimizations (such as making use of the 'support network' [3, Ch.7]).

associated to an unobserved RV (line 1 of RESAMPLE) or to an RV with an unobserved parent (line 5). As long as the specialized decision lists that we construct behave exactly the same with respect to this fixed set of CPD-queries as the original decision lists do, Gibbs sampling will indeed produce exactly the same samples with specialization as without.

There is a lot of existing work on transformation or specialization of logic programs that has the same end-goal as our work, namely transforming a given program to an "equivalent" but more efficient program [9]. However, we are not aware of any work that considers the same setting as we do, namely that of executing a fixed set of queries on a knowledge base with a static and a dynamic part, and specializing with respect to the static part. In particular, this setting makes our work different from the work on *partial deduction* for logic programs [6, 7]. In our setting, we know all input arguments of the queries but we know only part of the knowledge base on which they will be executed. In contrast, in the partial deduction setting, one knows only some of the input arguments of the queries but one knows the entire knowledge base. Hence, existing off-the-shelve systems for partial deduction (see e.g. Leuschel et al. [7]) are, as far as we see, not optimal for our setting.

Our specialization algorithm is shown in Figure 2. The main idea is the following. The CPD-predicates are defined in terms of the state predicates. The evidence is a partial interpretation of these state predicates (specifying the known state for a subset $\mathbf{O}$ of all concrete RVs). We want to specialize the definitions of the CPD-predicates with respect to this evidence. Since the evidence is defined at the ground level but the definitions of the CPD-predicates are at the non-ground level, we first (partially) ground these definitions before we specialize them. We now explain this further.

```
procedure SPECIALIZE(U, O, o)
1  for each CPD-predicate p
2      let D be the decision list for p
3      for each q ∈ AllQueries(p, U, O)
4          SPEC_DECISION_LIST(D, q, U, O, o)
```

```
procedure SPEC_DECISION_LIST(D, q, U, O, o)
1  if D is non-empty
2      let C be the first clause in D
       and D_rest be the other clauses in D
3      C_q = GROUND_HEAD(C, q)
4      let Head be the head and B_q the body of C_q
5      Body=SPECIALIZE_BODY(B_q, U, O, o)
6      if Body = true
7          ASSERT_FACT(Head)
8      else
9          if Body ≠ false
10             ASSERT_CLAUSE(Head, Body)
11         SPEC_DECISION_LIST(D_rest, q, U, O, o)
```

Figure 2: The specialization algorithm for the decision lists that define the CPD-predicates ($\mathbf{U}$ are the unobserved RVs, $\mathbf{O}$ the observed RVs and $\mathbf{o}$ their observed values).

The outer-loop of our algorithm (line 1 of the SPECIALIZE procedure in Figure 2) is over all the CPD-predicates: we specialize each CPD-predicate p in turn. To do so, we first collect all CPD-queries for p. As explained before, the only CPD-queries that we need are the ones associated to an RV that is unobserved or has an unobserved parent. The set of all such CPD-queries is denoted $AllQueries(p, \mathbf{U}, \mathbf{O})$ (line 3 of the SPECIALIZE procedure). We then loop over this set: for each CPD-query q we apply the SPEC_DECISION_LIST procedure. We explain this procedure by means of an example.

Example 5.1. Let p be *cpd_graduates*/2, let the decision list D that defines p be the same as given earlier in Example 3.1, and let the CPD-query q be $cpd\_graduates(s1, Distr)$. The SPEC_DECISION_LIST procedure starts by processing the first clause C in D:

```
cpd_graduates(S,[yes:0.2,no:0.8]) :- grade(S,_C,c), !.
```

First we ground the head variables of C with respect to q (line 3 of SPEC_DECISION_LIST) yielding the clause C_q:

```
cpd_graduates(s1,[yes:0.2,no:0.8]) :- grade(s1,_C,c), !.
```

Next, we apply the function SPECIALIZE_BODY to the body of C_q (line 5), yielding *Body* (see Example 5.2). There are three possible cases.

- If *Body* equals *true*, we assert a fact `cpd_graduates(s1,[yes:0.2,no:0.8])` (line 7). We can discard the remaining clauses in D with respect to q (these clauses will never be reached for q since only the first applicable clause in a decision list fires).
- If *Body* equals $false$, we discard C_q and continue with the next clause in D (line 11).
- Otherwise, we assert a clause of the form
  `cpd_graduates(s1,[yes:0.2,no:0.8]) :- Body, !.`
  (line 10) and we again continue with the next clause in D (line 11).

The function SPECIALIZE_BODY (Figure 2) is rather involved. For details we refer to the full paper [4]. We now give a very simple example.

Example 5.2. Let B_q, the body to be specialized, be `grade(s1,C,c)` (this is the situation of our previous example). First we ground the free variable C, yielding a disjunction B_1, namely `grade(s1,c1,c) ; ... ; grade(s1,cn,c)`. Then we specialize each of the literals in B_1 with respect to the evidence. Consider the first literal, `grade(s1,c1,c)`. If we have evidence that $s1$ obtained grade 'c' for course $c1$ then we replace the literal by *true*, if we have different evidence we replace it by $false$, if we have no evidence we leave it unchanged. Doing this for each literal yields a specialized disjunction B_2. Finally, we simplify B_2 using logical propagation rules (e.g. a disjunction is true if one if its disjuncts is true).

From the perspective of efficiency of the specialization process (time needed for specializing) our algorithm is not optimal: the specialization time can easily be reduced, for instance by more closely integrating the different steps of SPECIALIZE_BODY. However, in our experiments we observed that the specialization time is negligible as compared to the runtime of Gibbs sampling with the specialized decision lists (see the full paper [4]). Hence, we keep our specialization algorithm as simple as possible, rather than complicating it in order to reduce specialization time. This also makes it easier to see that specialization indeed preserves the semantics of the CPD-predicates (and hence that Gibbs sampling produces the same sequence of samples as without specialization).

6. Experiments

We now experimentally analyze the influence of specializing the definitions of the CPD-predicates on the efficiency of the Gibbs sampling algorithm.

We test our algorithms on three common real-world datasets: IMDB, UWCSE and WebKB. We obtained a parameterized Bayesian network for each dataset by means of machine learning. We use two inference scenarios. The first is *'prediction'*: there is one parameterized RV that we want to predict, all concrete RVs associated to that parameterized RV are unobserved, all others are observed. For each dataset we do multiple experiments,

each time with a different parameterized RV as the prediction target. The second scenario is *'missing data'*: a random fraction f of all concrete RVs is unobserved ('missing'), the others are observed. We use several values of f, ranging from 5% to 50%. For each value we repeat each experiment 5 times, each time with different unobserved RVs. We report the mean and standard deviation of the runtime across these 5 repetitions. More details about our experimental setup are given in the full paper [4].

We report the runtime of our Gibbs sampling algorithm in minutes. The *runtime without specialization* is the runtime of Gibbs sampling with parameterized CPDs that have not been grounded or specialized. The *runtime with specialization* is the sum of the specialization time and the runtime of Gibbs sampling with the specialized CPDs. Recall that both settings produce exactly the same sequence of samples.

The results for the 'missing data' scenario are shown in Figure 3. Using specialization always yields a speedup. The magnitude of the speedup of course greatly depends on the amount of evidence. On WebKB, the dataset that is by far the most computationally demanding, we get a speedup of an order of magnitude when there are 5% unobserved RVs. On the smaller datasets (IMDB and UWCSE), the speedups are more modest.

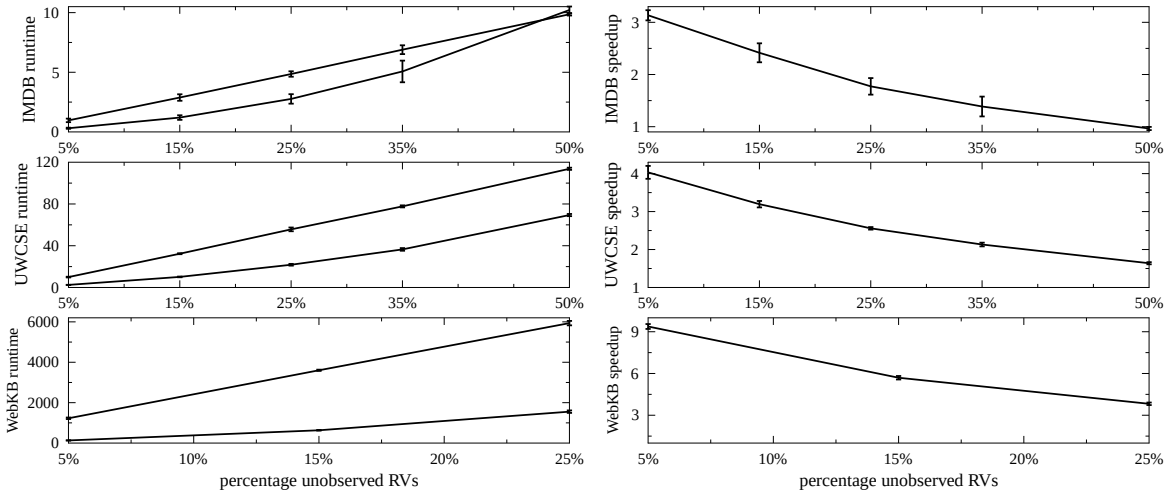

Figure 3: Results for the 'missing data' scenario. Left subgraphs show the runtime without (upper line) and with specialization (lower line); right subgraphs show the corresponding speedup-factor achieved due to specialization. Error bars indicate the standard deviation.

The results for the 'prediction' scenario are shown in Table 1. For half of the prediction targets, specialization yields significant speedups of a factor 4 to 7. For the other targets, the speedup is small to negligible (≤ 1.5). These are mostly cases where the state predicate that forms the computational bottleneck (e.g. because it is involved in a *findall*) is unobserved and hence cannot be specialized on.

In the above results (especially for the 'missing data' scenario), the speedups are the lowest on the smallest dataset (IMDB) and the highest on the largest one (WebKB). This suggest a correlation between the speedup due to specialization and the data-size. To investigate this, we performed additional experiments in which we varied the size of the datasets (see the full paper [4]). We found a clear trend: the larger the dataset, the higher

Table 1: Results for the 'prediction' scenario: runtime without specialization, runtime with specialization and speedup-factor achieved due to specialization.

Data/Target	No spec.	Spec.	Speedup	Data/Target	No spec.	Spec.	Speedup
IMDB/1	16.1	14.9	1.08	UWCSE/3	12.2	2.1	5.87
IMDB/2	2.6	1.7	1.51	UWCSE/4	71.8	15.8	4.55
UWCSE/1	75.1	17.4	4.31	WebKB/1	2628	406	6.48
UWCSE/2	10.9	10.4	1.05				

the speedup. This is a positive result: speedups are more necessary on large datasets than on small ones.

7. Conclusions

We considered the task of performing approximate probabilistic inference with probabilistic logical models by means of Gibbs sampling. We used the general framework of parameterized Bayesian networks. We showed how to represent the considered models and how to implement a Gibbs sampling algorithm for such models in Prolog. We argued that program specialization is suited to make this algorithm more efficient (which can in turn make the obtained inference answers more accurate) and introduced a concrete specialization algorithm. We experimentally investigated the influence of specialization on the efficiency of Gibbs sampling. Our results show that specialization yields speedups of up to an order of magnitude and that these speedups grow with the data-size.

References

[1] B. Bidyuk and R. Dechter. Cutset sampling for Bayesian networks. *Journal of Artificial Intelligence Research*, 28:1–48, 2007.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton. *Probabilistic Inductive Logic Programming*. Springer, 2008.

[4] D. Fierens. Improving the efficiency of Gibbs sampling for probabilistic logical models by means of program specialization. Technical Report CW-581, Department of Computer Science, Katholieke Universiteit Leuven, 2010. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW581.abs.html.

[5] D. Fierens, J. Ramon, M. Bruynooghe, and H. Blockeel. Learning directed probabilistic logical models: Ordering-search versus structure-search. *Annals of Mathematics and Artificial Intelligence*, 54(1):99–133, 2008.

[6] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4-5):461–515, 2002.

[7] M. Leuschel, S.J. Craig, M. Bruynooghe, and W. Vanhoof. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*, volume 3094 of *Lecture Notes in Computer Science*, pages 340–375. Springer, 2004.

[8] R.E. Neapolitan. *Learning Bayesian Networks*. Prentice Hall, New Jersey, 2003.

[9] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19-20:261–320, 1994.

[10] D. Poole. First-order probabilistic inference. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 1997)*, pages 985–991. Morgan Kaufmann, 2003.

[11] V. Santos Costa. On the implementation of the CLP(BN) language. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, volume 5937 of *Lecture Notes in Artificial Intelligence*, pages 234–248. Springer, 2010.